Lesson 04

CSC140 Foundations of Computer Science

17 February 2020

What we have learned thus far.

Variables

A variable is a named location in the computer's memory.

A variable has six attributes:

1. name—our style guide recommends descriptive names constructed by connecting words with underscores and for the use of lower case letters (e.g., sales_tax rather than st)
2. type
3. value
4. location (address)—we will neither specify nor examine the addresses of our variables, but will instead let the interpreter manage addresses for us
5. scope—in which part of the source code is the variable defined? (answers the question: where?)
6. lifetime—during which part of the program's execution is the variable defined? (answers the question: when?)

Types

The type of a variable determines:

- the amount of memory that the variable requires
- how the variable's value is encoded in the computer's memory
- which operations can be used to alter the variable's value or combine the value with other values

We have already seen the **int**, **float**, and @**str** types.

You can guess what the **bool** type is. Enter this code in the interpreter:

```
2 + 2 == 4
type(2 + 2 == 4)
9 < 8
type(9 < 8)
```

What is the **bool** type?

These types (with some small changes in spelling) are common to most programming languages.

Other popular programming languages place a limit on the magnitude of an integer that can be stored in an **int** variable. This is not the case in Python.

Internally, the interpreter represents **float** (floating point) variables with scientific notation. A student in a chemistry writes Avogrado's number as $6.022 \cdot 10^{23}$. (That's a lot better than writing 6 followed by 23 more digits!) The 6.022 is the mantissa. The 23 is the exponent. Similarly, the representation of a floating point value in the computer's memory contains a mantissa and an exponent.

Each integer value is an exact value. Floating point variables give programmers a way to represent real numbers approximately. Because there are an infinite number of real numbers between any two given real numbers, it is not possible to represent all real numbers exactly with mantissas whose size must necessarily be finite.

Operators

A programming language provides arithmetic, relational, and logical operators. Here are Python's opertors:

arithmetic + − * / // % **

relational < <= == != >= >

logical **and or not**

The meaning of + is overloaded. This means it has different meanings in different contexts.

Try this code:

```
17 + 2
"straw" + "berry"
```

What are two different meanings of +?

The logical operators can be defined with truth tables:

A	**B**	**A or B**
T	T	T
T	F	T
F	T	T
F	F	F

A	**B**	**A and B**
T	T	T
T	F	F
F	T	F
F	F	F

A	**not A**
T	F
F	T

Python differs from some other popular languages by including an operator (**) for exponentiation. It also includes distinct operators (/ //) for integer and floating point division. Other popular languages also distinguish between integer and floating point divisions, but do so differently. Python's logical operators are words. In some other languages, the logical operators are represented by characters that are not letters.

Statements

With variables, literals (numbers, strings, False and True), and operators, we can compose expressions.

```
(1 - t) * a + t * b
```

With a variable and an expression, we can compose an assignment statement.

```
weighted_average = (1 - t) * a + t * b
```

With relational and logical operators, we can compose an **if** statement.

```
if 0.0 < t and t < 1.0:
    weighted_average = (1 - t) * a + t * b
```

Python allows a more concise version of this condition. (Some other popular programming languages do not allow this shortcut.)

```
if 0.0 < t < 1.0:
    weighted_average = (1 - t) * a + t * b
```

What if the weight for the weighted average does not have a sensible value?

```
if 0.0 < t < 1.0:
    weighted_average = (1 - t) * a + t * b
else:
    print( "The value of t must lie between 0.0 and 1.0." )
```

What if we want evenly spaced values between two bounds?

```
# print 2.0, 2.2, 2.4, 2.6, and so on up to 3.8
a = 2
b = 4
for i in range(10):
    fraction = i / 10
    value = (1 - fraction) * a + fraction * b
    print( value )
```

Statements give programmers the means to...

- compute, store, and retrieve values
- instruct the computer to execute a series of actions
- instruct the computer to choose between (or among) alternative actions
- instruct the computer to execute an action repeatedly

What we will learn next

Compound data types

We will learn to bundle related values. We can create variables with contain several related values.

```
# define a list of people who have won the Turing Award
turing_award_winners = ['Brooks', 'Dijkstra', 'Knuth']
# each element of a list has an index
print( turing_award_winners[0] )
print( turing_award_winners[1] )
print( turing_award_winners[2] )

# a programmer can change the value of an element
# of a list
turing_award_winners[0] = "Frederick P. Brooks, Jr."
```

```
print( turing_award_winners[0] )

# define a tuple that contains a first name and last name
computer_scientist = ('Donald', 'Knuth')
# each element of tuple has an index
print( "first name = ", computer_scientist[0] )
print( "last name = ", computer_scientist[1] )

# tuples are immutable——it is not possible to change
# the value of an element

# define a dictionary
professor = { 'name': 'Donald Knuth',
              'institution': 'Stanford University' }
# each element of a dictionary has a key and a value
# here, use a key to print a value
print( professor["name"] )
print( professor["institution"] )
```

We will need to learn more about loops.

```
for i in range( len(turing_award_winners) ):
    print( turing_award_winners[i] )

for n in computer_scientist:
    print( n )

for k in professor.keys():
    print( professor[k] )

for n in professor.values():
    print( n )
```

Functions

We can also bundle related statements. We will begin by bundling related statements in functions.

```
def fahrenheit_to_celsius( temperature ):
    temperature = temperature - 32
    temperature = temperature * 5 / 9
    return temperature
```

This gives us a means of writing more concisely. We define a function and then can call (use) it many times:

```
freezing_point = fahrenheit_to_celsius( 32 )
comfortable_temperature = fahrenheit_to_celsius( 68 )
boiling_point = fahrenheit_to_celsius( 212 )
```

Use a five step recipe when writing a function:

- State the purpose of the function in a single, short, simple sentence: "Given a temperature on the Fahrenheit scale, return the equivalent temperature on the Celsius scale."
- Give the function a name that clearly indicates its purpose: fahrenheit_to_celsius
- Specify the number and type of parameters: "the caller of the function must provide a single floating point value that represents a temperature on the Fahrenheit scale."
- Specify the type of value that the function will return to its caller: "the function returns a floating point value that represents a temperature on the Celsius scale."
- Compose a sequence of arithmetic and logical operations that computes the function's return value.

  ```
  temperature = temperature - 32
  temperature = temperature * 5 / 9
  return temperature
  ```

Resist the temptation to jump straight to the last step!

If you do not get the first four steps right before you start on the last step, you will waste a lot of time.

If at the outset you do not know how to compute the desired result, you can still accomplish much by writing a *stub function*:

```
def fahrenheit_to_celsius( temperature ):
    return 0.0
```

This is syntactically correct Python. The interpreter will parse and execute this code. Of course, the function will return 0.0 to its caller every time and so will return a correct answer only when *temperature* = 32.

To write a stub function, complete the first four steps of the recipe for writing functions, then write a statement that returns a fixed value (maybe 0 for a function that returns an **int** value or False for a function that returns a **bool** value). This value can be arbitrary. It is just a place holder.

Programmers write stub functions so that they can work on other parts of a program until they find (for example, by reading online or by asking a teammate) the algorithms that they need to complete those functions.

Exercises

Practice writing functions

Write a program that defines and tests functions that...

- compute the arithmetic mean of two numbers
- compute the geometric mean of two numbers
- compute the harmonic mean of two numbers
- computes the Euclidean distance between two points in the plane, each of which is represented by a tuple
- computes the Manhattan distance between two points in the plane, each of which is represented by a tuple

Use this template:

```
import math

# write definitions of functions here

if __name__ == '__main__':
  # write calls to functions here
```

Use your new knowledge of functions to draw a landscape

Write a program that defines functions that draw elements of a landscape. Use the Turtle module. You might, for example, write functions that...

- draw a house
- draw a window or a door
- draw a brick wall
- draw a tree
- draw a car